1) We return unsorted elements when we call quicksort on an array with less than k elements.

∴ No. of subarrays is approximately n/k

∴ The level will be $\log(n/k)$.

∴ Runtime of the quicksort part will be $O(\log(n/k) \times n)$

Now we have $\frac{n}{k}$ subarrays with k elements each.

Insertion sort will take $\frac{k}{n} \times O(k^2)$ to sort, which can be simplified as $O(nk)$

∴ $\boxed{\text{Total runtime is } O(nk + n\log(n/k))}$

We need to find a strategy for selecting k, such that runtime of quicksort is greater than the proposed algorithm of a combination of quick sort and insertion sort.

Since running time of normal quick sort is nlogn, we get

$q \cdot n \log n > i \cdot nk + q \cdot n \log(n/k)$

where q is the constant time for quicksort while i is the constant for insertion sort.

$\therefore q \cdot \log n > i \cdot k + q \cdot \log n - q \log k$

$\therefore q \log k > i \cdot k$

$$\therefore \boxed{\frac{q}{i} \log k > k}$$

These constants can only be found out by trial and error.

Q2

a) If all the elements are equal, the partition algorithm will always return a large subarray of $(r-1)$ elements and the pivot r itself.

$\therefore$ We get the recurrence,

$$T(n) = T(n-1) + \theta(n)$$
$$= \theta(n^2)$$

$\therefore$ It will behave exactly the same as a normal quicksort and we know that this is the worst case.

b)

```
PARTITION'(A, p, r)
{
    pivot = A[p]
    q = p
    t = p
    for i = p+1 to r
        if A[i] < pivot
            temp = A[i]
            A[i] = A[t+1]
            A[t+1] = A[q]
            A[q] = temp
            increment q and t
    else if A[i] == pivot
            swap A[i] and ~~A[t]~~ A[t+1]
            increment t
    return (q, t)
}
```

c)

```
QUICKSORT'(A, p, r)
{
    if p < r
        (q, t) = RANDOMIZED-PARTITION(A, p, r)
        QUICKSORT'(A, p, q-1)
        QUICKSORT'(A, t+1, r)
```

d) The analysis remains the same as the subarrays returned by the partition algorithm do not include the pivot in them. Hence, the assumption that all elements will be distinct still holds in the sense that a pivot value will never be repeated.

Homework 5

Q3]

$A = [7, 1, 8, 2, 3, 6, 4, 1, 5, 3]$

Max number is 8.

Auxillary array

Index	1	2	3	4	5	6	7	8
value	2	1	2	1	1	1	1	1

Taking cumulative sum,

Index	1	2	3	4	5	6	7	8
value	2	3	5	6	7	8	9	10

Shifting to the right by 1,

1	2	3	4	5	6	7	8
0	2	3	5	6	7	8	9

Let B be the output array and C be the array having the cumulative sum.
Decrementing the value in C as we traverse B backwards =>

∴ B: Index	1	2	3	4	5	6	7	8	9	10
value					3					

C: Index	1	2	3	4	5	6	7	8
value	2	3	4	6	7	8	9	10

Traversing the original array from right to left, adding elements into the output array ~~on~~ one by one:→

⇒

B: Index	1	2	3	4	5	6	7	8	9	10
value					3		5			

Index	1	2	3	4	5	6	7	8
value	2	3	4	6	6	8	9	10

⇒

B Index	1	2	3	4	5	6	7	8	9	10
value		1			3		5			

Index	1	2	3	4	5	6	7	8
value	1	3	4	6	6	8	9	10

⇒

B Index	1	2	3	4	5	6	7	8	9	10
Value		1			3	4	5			

Index	1	2	3	4	5	6	7	8
Value	1	3	4	5	6	8	9	10

∴ Similarly, ~~for~~ after performing the same for remaining elements

B: Index	1	2	3	4	5	6	7	8	9	10
value	1	1	2	3	3	4	5	6	7	8

C: Index	1	2	3	4	5	6	7	8
Value	0	2	3	5	6	7	8	9

Q4]

- As per the hint, initially, all the elements are potentially maximum or minimum.
- We maintain two separate arrays for minimum and maximum
- First, we compare sets of two elements each from the main array. We add the smaller number in the array maintaining the potentially minimum numbers array while we add the greater number in the potentially maximum numbers array.
- Thus, after the previous step, we will have two arrays of n/2 length each. And the total number of comparisons made will also be **n/2. (One comparison per two elements.)**
- After we have separated the potentially max and min elements into two parts, we can evaluate minimum and maximum numbers in the array holding potentially minimum numbers and the array holding potentially maximum numbers respectively by using any comparison sorting algorithm, we know that the total number of comparisons made will be **n/2-2 for each array.**
- Thus, the total number of comparisons further to find the min and max from their respective arrays will be n-2.
- Adding n/2 from the first step and then n-2 comparisons from the consecutive steps, we get **(3n/2) – 2** comparisons.

Q5]

a) Sorting will take O(nlog(n)) time (using merge sort) and displaying the ith largest will take a constant time “i”. Therefore total running time will be O(**n*log(n) + i)**

b)

a. Building a max-priority queue will take O(**nlog(n))** (heapify) time and we know that EXTRACT-MAX takes O(log(n)) time. Therefore, performing EXTRACT-MAX i times will take O(**i*log(n))** time.
b. Thus the total running time will be O(**n*log(n) + i*log(n))**
c. **Running time = O((n+i)log(n))**

c)

a. Partitioning around the ith largest number takes running time of O(n) (Via Randomized Selection Algorithm)
b. Sorting the i largest elements after the partition will take O(i*log(n)) time (using merge sort)
c. Therefore running time = **O(n + i*log(n))**